Representing Multidimensional Trees

Earlham Theory Group

April 9, 2006

1 Introduction

Traditionally, the context-free grammars (CFGs) are represented as a set of string rewriting rules: a CFG is a four-tuple $\mathcal{G} = \langle \Sigma, V, S, P \rangle$, where Σ is the terminal alphabet, V is a finite set of non-terminal symbols, $S \in V$ is the initial symbol, and P is a finite set of productions—each of which map some symbol, $x \in V$, to a string of symbols, $y \in (\Sigma \cup V)^*$ [Sip01].

We often are interested in the parse trees that derive strings in a grammar, rather than just the strings that are derivable. We can choose to interpret the definition of $\mathcal{G}$ differently, and represent the context-free grammars using local trees [GS84, Rog03]. Interpret each production in P as a local tree (a tree with height $h \leq 1$) having a yield labeled in $(\Sigma \cup V)^*$ and a root labeled in V. Let each member of Σ label the root of a trivial local tree, and let the initial symbol S label the root of a trivial derivation tree T_0. The context-free derivation of any T_{i+1} from T_i can be performed by by replacing a leaf node of T_i, labeled by some x, with a local tree in P that has root labeled x—that is, by concatenating local trees from P to the derivation tree T_i. The derived string, at any point in the process, is the string yield (the left-to-right concatenation of the leaf nodes) of T_i. A string is in the language recognized by $\mathcal{G}$, $L(\mathcal{G})$, if and only if it is in Σ^* and is the string yield of some derivation tree.

We can see that, in this representation, each local tree in P is grammatically equivalent to a production in a traditional CFG—the tree maps some root symbol, $x \in V$, to some ordered set of children $y \in (\Sigma \cup V)^*$. Figure 1 shows an example grammar in local tree form, plus an example derivation tree constructed from rules in the grammar. The example derivation tree yields the string (())()().

Tree adjoining grammars (TAG) [JS96] lift the context-free grammars from a string-generating formalism to a tree-generating formalism. Just as context-free grammars can be represented as a set of string rewriting rules,

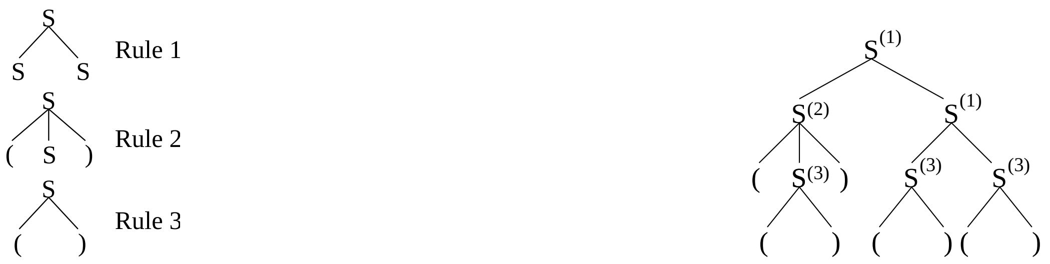

Figure 1: An example context-free grammar in local tree form, plus one possible derivation tree with root labeled by S.

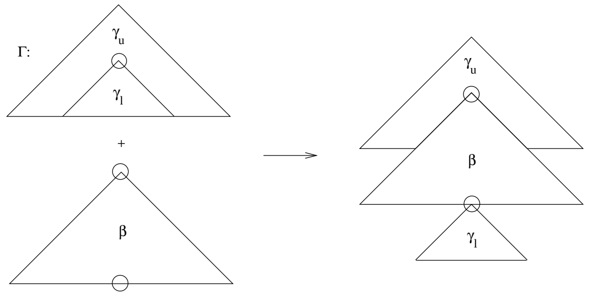

Figure 2: Adjunction of tree β into Γ_i.

tree adjoining grammars can be represented as a set of tree rewriting rules. While CFGs derive strings by symbol replacement, TAGs derive trees by a process known as adjunction.

Conceptually, tree adjunction in TAG is the substitution of one tree into another at some node (Figure 2). To adjoin some auxiliary tree β to some derivation tree Γ_i at node x, we require that the auxiliary tree β have a root node with the same label as x, as well as a distinguished leaf node with the same label as x (known as the "foot" node, or foot(β)). To perform the adjunction, we create two trees from Γ_i—γ_u (identical to Γ_i, but with the subtree below x deleted), and γ_l (the subtree rooted by x). We create Γ_{i+1} from Γ_i via two replacements: x in the frontier of γ_u is replaced by β, and foot(β) is replaced by γ_l.

Adjunction, if allowed to occur only at frontier nodes, is exactly equivalent to tree concatenation. The ability to replace non-frontier nodes with trees is what differentiates TAGs from our local tree representation of CFGs. This same ability affords TAGs a degree of context-sensitive generative power not found in CFGs.

Thus far, we have defined a local tree as a traditional parent/child n-branching tree with restricted height ($h \leq 1$). From a different point of view, a local tree is of the exact structure of a single CFG production: it maps

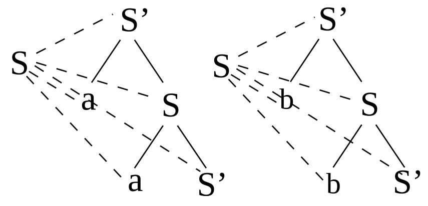

Figure 3: Two three-dimensional local trees. These trees denote a tree-adjoining grammar in local tree form.

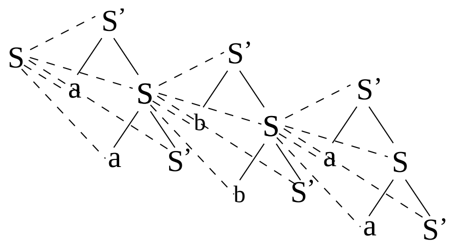

Figure 4: One possible three-dimensional derivation tree.

one point (the root symbol) to one string of symbols licensed to replace it. From this perspective, a local tree can be thought of as an arbitrary one-dimensional string of symbols, dominated in the second dimension by exactly one symbol. This definition of a local tree can be generalized to arbitrary dimensions, in a way analogous to the construction of topological simplicies. To construct a d-dimensional local tree T^d from an arbitrary $(d-1)$-dimensional tree T^{d-1}, we add exactly one new node. This new node serves as the root of T^d and is the immediate predecessor, in an orthogonal dimension[1], of every node in T^{d-1} (Figure 3). The local yield of T^d is T^{d-1}.

Concatenation of these local d-dimensional structures allows the construction of arbitrary d-dimensional trees (Figure 4). For an arbitrary d-dimensional tree, we note that the $(d-1)$-dimensional yield is constructible: the yield of an arbitrary d-dimensional tree is obtained by combining, in some well-defined order[2], all of the $(d-1)$-dimensional yields of its d-dimensional local components (Figure 5).

For the CFGs, we were able to represent string rewriting via concatenation of trees—that is, we were able to reduce substitution in a structure to con-

[1]To represent a multi-dimensional local structure on a two-dimensional medium, we place each d-dimensional root to the left of the $(d-1)$-dimensional structure it dominates, and connect it to each node in that $(d-1)$-dimensional structure with a line. A unique line style is used for each dimension.

[2]For dimensions greater than two, there is an ordering ambiguity that necessitates the storage of extra per-node information.

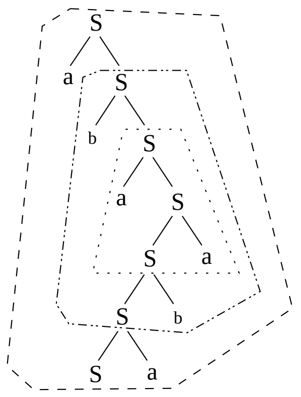

Figure 5: The two-dimensional yield of a three-dimensional derivation tree.

catenation of structures—purely by raising the dimensionality of the structure. We moved from substitution on one-dimensional structures (strings) to concatenation of two-dimensional local structures, while maintaining equivalent generative power. In a similar fashion, we can achieve generative power equivalent to that of TAG using only concatenation—by moving from adjunction and substitution in two-dimensional structures to concatenation of three-dimensional local structures [Rog03].

More simply, we can capture the generative power of TAG with a slight variation on our definition of CFGs as sets of local structures: we need only raise the dimensionality of the local structures in P by one. This process may be iterated to higher dimensions, creating an infinite hierarchy of language classes that, in terms of string-generating power, directly corresponds to Weir's control language hierarchy [Wei92].

If viewed as a multidimensional grammar, Figure 3 captures the copy language over $\{a, b\}$ using a set of three-dimensional local trees. Figure 4 shows one possible derivation in this grammar. Figure 5 shows the two-dimensional yield of the derivation tree in Figure 4, with circles around the two-dimensional yield of each three-dimensional local tree. The one-dimensional yield of Figure 5 is the string recognized by the derivation tree in Figure 4.

Dewdney [Dew74] discusses a similar (but formally distinct) graph-theoretic notion of multidimensional trees. A formally equivalent notion of multidimensional trees, as well as a formally distinct notion of multidimensional grammars, is discussed by Baldwin and Strawn [BS91]. Adaptive k-d trees [Sed98], while useful for searching in a multidimensional space, are not strictly multidimensional in their structure.

Our group is interested in building parsers for grammars based on multidimensional trees. In this paper, we develop a formal definition of these multidimensional trees as abstract structures that can be directly realized as an ADT. We then implement this ADT as a C++ class. Further, by interpreting this abstract structure algebraically we obtain a flat form suitable for file input and output.

2 Tree Construction

2.1 Tree-building Operations

The classic representation of 2-dimensional trees, a parent with a set of children, is difficult to generalize into a form for a tree in higher dimensions. Since a 2-dimensional tree has arbitrarily many 2-dimensional children that have a 1-dimensional ordering, a node in a 3-dimensional tree would have arbitrarily many 3-dimensional children that have a 2-dimensional (partial) ordering. For an n-dimensional tree, a node would have arbitrarily many n-dimensional children with an $(n-1)$-dimensional (partial) ordering.

Instead, we choose to use a well-known "left child, right sibling" representation [Sed98] for 2-dimensional trees. Instead of maintaining references to arbitrarily many 2-dimensional children, each node contains a reference to a single minimum 2-dimensional successor (the "left child") and to a single minimum 1-dimensional successor (the "right sibling"). With this representation, the first child of a node is positioned as the 2-dimensional successor of that node. The remaining children are then each placed as the 1-dimensional successor of the preceding child. Note that in full generality, this representation admits the possibility that the root of the tree may have a right sibling. We will interpret such a structure as a linearly ordered forest, and we will interpret the case where the root has no right sibling as the trivial forest, a single tree.

Bottom-up construction for the usual left-child/right-sibling trees has two operations as seen in Figure 6: extending the forest by adding another tree at the beginning of the ordered forest (which we will call ExLeft), and adding a root as the parent of the trees in an ordered forest (which we will call ExUp).

Definition 1 *Linearly Ordered Forests*

- $\sim$ *is an empty forest.*
- *If* t_1 *is a tree and* t_2 *is a linearly ordered forest then* ExLeft(t_1, t_2) *is a linearly ordered forest.*

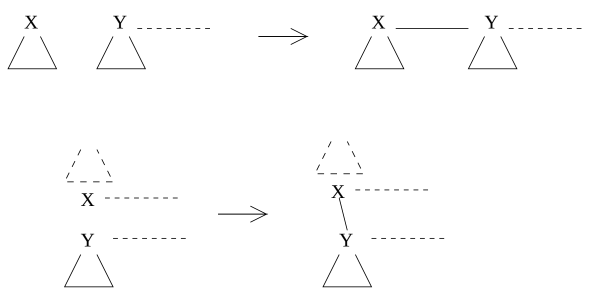

Figure 6: The two tree-building operations.

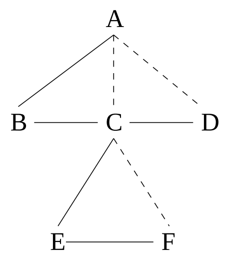

Figure 7: An example two-dimensional tree.

- *If* t_1 *is a linearly ordered forest and* $X \in \Sigma$ *then* ExUp(X, t_1) *is a tree.*
- *Nothing else is a linearly ordered forest.*

This allows us to construct arbitrarily branching 2-dimensional trees using only two links per node.[3] We will later generalize this structure to allow us to construct arbitrary dimensional, arbitrary branching structures requiring only one link per dimension per node.

2.1.1 Example

First we will look at a 2-dimensional example using these two tree building operations. We will build the tree shown in Figure 7. In the figure, solid lines represent the actual links, while the dotted lines are there only to better visualize the tree structure. An annotated version of the tree in Figure 8 shows the various local trees and forests that make up the tree as circled groups.

Starting at the bottom, ExUp$(F, \sim)$ creates tree $t1$. ExLeft$(\sim, t1)$ combines $t1$ with an empty tree to form the singleton forest $f1$. Similarly, ExUp$(E, \sim)$ forms tree $t2$, which is then added to $f1$ with ExLeft$(f1, t1)$

[3]An interesting consequence of using the left-child/right-sibling tree organization is that it allows every n-branching tree to be embedded in a binary branching structure.

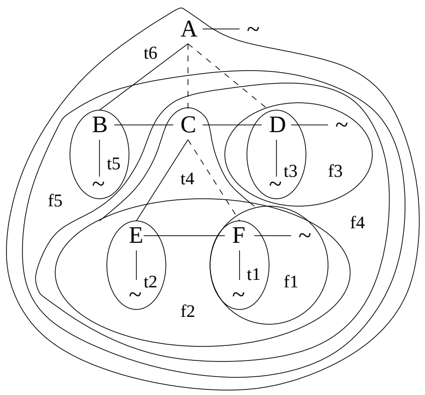

Figure 8: An example tree annotated to show trees and forests.

to create $f2$. Repeating this step once again, $\textsc{ExUp}(D, \sim)$ creates $t3$ and $\textsc{ExLeft}(\sim, t3)$ creates $f3$. $t4$ is created with $\textsc{ExUp}(C, f2)$ and then added to $f3$ to create $f4$ through $\textsc{ExLeft}(f3, t4)$. $\textsc{ExUp}(B, \sim)$ creates $t5$ which is used in $\textsc{ExLeft}(f4, t5)$ to create $f5$. The tree is then completed with $\textsc{ExUp}(A, f5)$ to create $t6$ and $\textsc{ExLeft}(\sim, t6)$ to construct the final tree.

2.2 Unified Constructor

Generalizing the two tree-building operations to arbitrary dimensions is simpler if we create a new, unified constructor. Instead of first adding siblings to create a forest, and then assigning the forest a root node, we will do this in one step. Taking a label and two forests, we will create a new node that is the parent of the roots of the trees in the first forest, and is the leftmost sibling of the second forest. The "root of a forest" is simply the root of the minimum tree in that forest. In contrast, we may refer to the minima of a forest—these are simply the roots of the individual trees of the forest, and they are incomparable to each other in the major dimension of the forest. The root of the forest is the unique minimum of the forest if and only if the forest is a singleton forest, i.e. a tree.

Using the unified constructor, a 2-dimensional (singleton) forest is formed if the forest has no 1-dimensional child. Otherwise, a proper forest is formed. Each of the siblings of the new node is a minimum.

Definition 2 *Linearly Ordered Forests—Unified Constructor*

- $\sim$ *is an empty linearly ordered forest.*
- *If* t_1 *and* t_2 *are linearly ordered forests and* $X \in \Sigma$ *then* $T(X, t_1, t_2)$ *is*

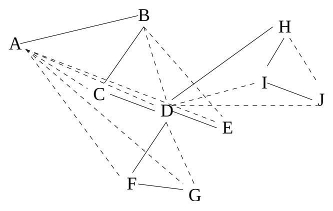

Figure 9: An example three-dimensional tree.

a linearly-ordered forest. Here t_2 is the set of two-dimensional children of the new node labeled X, and t_1 is the set of its siblings to the right.

- *Nothing else is a linearly ordered forest.*

2.2.1 Example

Let us use the unified constructor to construct the tree in Figure 7. Every pair of ExUp and ExLeft invocations that use the same first argument can be combined, with that shared term as the first argument of the T constructor. The remaining arguments of the constructor would be the remaining terms of the ExUp and ExLeft instances, respectively. The terms of the construction would be:

$T(F, \sim, \sim) = f1$
$T(E, f1, \sim) = f2$
$T(D, \sim, \sim) = f3$
$T(C, f3, f2) = f4$
$T(B, f4, \sim) = f5$

$T(A, \sim, f5)$ completes the tree.

2.3 Extending to Arbitrary Dimensions

As our trees become more complex, it is helpful to define some terminology. This terminology should allow us to talk about various aspects of an n-dimensional tree without our description becoming obfuscated in bulky explanation.

Definition 3 *n-dimensional Local Tree*
A local tree is a tree of height ≤ 1 in each dimension. This consists of a root and an $(n-1)$-dimensional yield, which we will call the local yield.

Definition 4 *$(n-1)$-dimensional Local Yield*

An $(n-1)$-dimensional local yield of a node is the set of all n-dimensional children of that node, which are ordered as an $(n-1)$-dimensional singleton forest, a tree—there must be exactly one minimum with respect to the $(n-1)$-dimensional ordering.

Definition 5 *$(n-1)$-dimensional Child Structure*

An $(n-1)$-dimensional child structure of an n-dimensional node is the forest of trees rooted at that node's $(n-1)$-dimensional local yield.

Note that the local yield of a node is an $(n-1)$-dimensional tree of nodes, while the child structure of the same node is an $(n-1)$-dimensionally ordered forest of n-dimensional trees rooted at those nodes.

Now we can restate the way our 2-dimensional trees are built. Every node in a 2-dimensional forest roots two local trees: a 2-dimensional local tree, the yield of which are the node's two-dimensional children, and a 1-dimensional local tree whose yield is the node's right sibling. Consequently, the 2-dimensional successor is the minimum point in the node's 1-dimensional child structure, and the 1-dimensional successor is the minimum point in the node's 0-dimensional child structure. Both these child structures are linearly ordered forests. To extend this to arbitrary dimensions, we will say that every node in an n-dimensional tree has n successors. Each successor in the ith dimension is the minimum point in that node's $(i-1)$-dimensional child structure for $1 \leq i \leq n$. We extend the unified construction as follows:

Definition 6 *(Preliminary) Tree-ordered Forests*

- *$\sim$ is an (empty) d-dimensional forest.*
- *If $t_1, t_2, \ldots, t_d$ are tree ordered forests and $X \in \Sigma$ then $T(X, t_1, t_2, \ldots, t_d)$ is a tree-ordered forest. t_i is the set of i-dimensional children of the new node labeled X.*
- *Nothing else is a tree-ordered forest.*

2.3.1 Example

To see the unified constructor for n dimensions, we will look at the 3-dimensional tree shown in Figure 9. As in Figure 7, the solid lines are the actual links while the dotted lines are present to better visualize the tree structure. Again, the tree has been annotated (Figure 10) to show the local trees and forests.

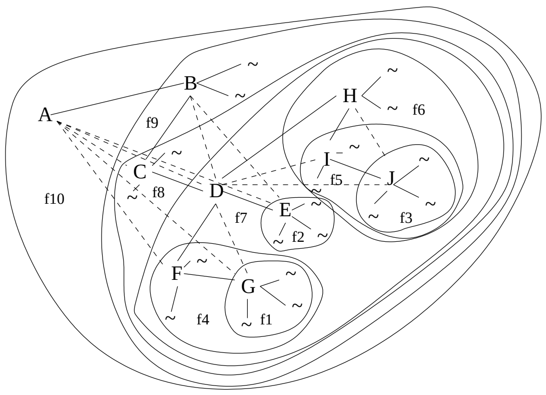

Figure 10: An example three-dimensional tree, annotated to show trees and forests.

The constructor for this tree will take four arguments: the label of the node, and three forests. We will indicate this construct as $T(W, X, Y, Z)$ where W is the label, X is a forest and the 1-dimensional successor of W, Y is a forest and the 2-dimensional successor of W, and Z is a forest and the 3-dimensional successor of W. The empty tree is represented by $\sim$. In all of our constructions of trees, the order that the forests are constructed in is flexible. Until two forests are combined, they can be built independently of one another.

First we will construct $f6$:
$T(J, \sim, \sim, \sim) = f3$
$T(I, f3, \sim, \sim) = f5$
$T(H, \sim, f5, \sim) = f6$

Then we create $f4$:
$T(G, \sim, \sim, \sim) = f1$
$T(F, f1, \sim, \sim) = f4$

And the last child of D is $f2$:
$T(E, \sim, \sim, \sim) = f2$

Next we combine them in D:
$T(D, f2, f4, f6) = f7$

And then we can continue to build the rest of the tree:
$T(C, f7, \sim, \sim) = f8$

$T(B, \sim, f8, \sim) = f9$
$T(A, \sim, \sim, f9) = f10$

2.4 (i, d)-forests

Our definition of tree-ordered forests is not yet complete. In the above example, note that when assigning $f6$ as the 3-dimensional successor of D, the root of $f6$ does not have a 1-dimensional successor. In fact, it *can't* have a 1-dimensional successor—it is the root of a 2-dimensional local yield, which by our definition must be a singleton 2-dimensional forest. Only the empty tree can be used for this constructor argument, and Definition 6 needs to be modified to include this. To denote a d-dimensional forest with a unique minimum (within a local structure, in this sense) in dimension i, we define an (i, d)-forest, where $0 \leq i \leq d$, as a forest whose root has an empty j-dimensional local yield for all $j < i$. A node with no i-dimensional children for all $i < j$ can be interpreted as the root of a (j, d)-forest. Having done this, we can insist that the only forests accepted as a 3-dimensional successor in the unified constructor be $(2, d)$-forests, the subset of forests that do not have a 1-dimensional successor.

Lemma 1 *If X is the root of an i-dimensional local yield, it has no j-dimensional successor for $j < i$.*

Proof: When $i = 0$ or $i = 1$, this is trivially true. For $i > 1$, if X has any j-dimensional successor for $j < i$, then it is not the unique minimum of the local yield and hence is not the root of that local yield, which must be a singleton forest. $\dashv$

Corollary 1 *Every (i, d)-forest is also a (j, d)-forest for $j < i$.*

Definition 7 *Tree-ordered Forests—Fully Typed*

- $\sim$ *is an (empty) (i, d)-forest for all $0 \leq i \leq d$*
- *If t_1, t_2, $\ldots$, t_d are, respectively, $(0, d)$-, $(1, d)$-, $\ldots$, $(d-1, d)$-forests and $X \in \Sigma$ then $T(X, t_1, t_2, \ldots, t_d)$ is a (j, d)-forest for all $0 \leq j \leq i$, where i is the smallest dimension such that t_i is not empty, or d if all t_k are empty. Here each t_k is the successor of the new node labeled X in the kth dimension.*
- *Nothing else is a tree-ordered forest.*

By Corollary 1, the set of (i,d)-forests is a subset of the set of $(i-1,d)$-forests for all $0 < i \leq d$.

Going back to Figure 10, we can label each tree as an (i,d)-forest. $f1$, $f2$, and $f3$ are (0,3)-forests. $f4$, $f5$, $f7$, and $f8$ are (1,3)-forests and, by Corollary 1, (0,3)-forests. $f6$ and $f9$ are (2,3)-forests (and both (1,3)- and (0,3)-forests). Finally, $f10$ is a (3,3)-forest.

3 Concrete Forms

In order to work with these trees in more concrete applications such as programs, we must develop a realization of these trees both as an abstract data type for storing trees in memory, and as a string to store trees in files and use as input.

3.1 Abstract Data Type

In order to create programs to work with these trees, we will need a way of constructing data structures that can represent an n-dimensional forest. The information we need to store falls directly out of the definition of forests. The forest constructor T takes a label and a sequence of (i,d)-forests. Our data structure will then store the label of the node created and references to the forests that are that node's local yields. With this model, one instance of our structure can represent one node, with the label and pointers to the local yields being stored. Since each local yield is made up of smaller local yields, what is really being pointed to is a node that is the minimum of that local yield.

The constructor for building forests follows Definition 7. This function takes a list of references to forests and a label and returns a new forest rooted at that label. Each element of the list of forests passed to the constructor represents its successor in a particular dimension designated by the ordering.

Selectors would be needed to determine the label and successor of each node. Mutators could also be included to allow modification of the label and the successors, but this is not necessarily needed since the constructor can be used to set these.

Here is an example of this ADT, realized as a C++ class:

```
template<class label_type>
class forest
{
 public:
  forest(label_type label,
         vector<forest*> links)
  void set_label(label_type new_label);
  void set_link(std::size_t link_number,
                forest* new_link);

  label_type get_label( ) const;
  tree* get_successor(
        std::size_t link_number) const;

 private:
  label_type label;
  vector<forest*> link;
}
```

Because our class uses the the C++ `vector` in its implementation, it is truly polymorphic in its dimensionality. The dimensionality of the forests is not explicit in this data type; it is however noted in the size of the vector. While the constructor allows for a d-dimensional forest to reference forests of different dimensionality this may be disallowed by adding assertions to the constructor to ensure uniformly dimensional trees.

3.2 Flat Form

When working with complex trees as the input or output of a program or attempting to store trees in text files, it becomes necessary to develop a concise way of representing the tree in a flat (string) form. A form of this type can be taken directly from the free algebra generated by the constructor from Σ and $\sim$. One variation, to allow for more concise and easier to read trees, is to write the terms as $X(t_1, t_2, \ldots, t_d)$ rather than $T(X, t_1, t_2, \ldots, t_d)$.

Definition 8 *Flat Form*

- $\sim$ *is an (empty)* (i, d)*-forest in flat form for all* $0 \le i \le d$
- *If* t_1*,* t_2*, ...,* t_d *are, respectively,* $(0, d)$*-,* $(1, d)$*-, ...,* $(d-1, d)$*-forests in flat form and* $X \in \Sigma$ *then* $X(t_1, t_2, \ldots, t_d)$ *is a* (j, d)*-forest in flat form for all* $0 \le j \le i$*, where* i *is the smallest dimension such that* t_i *is not empty, or* d *if all* t_k *are empty. Here each* t_k *is the successor of the new node labeled* X *in the* k*th dimension.*
- *Nothing else is a forest in flat form.*

3.2.1 Example

The terms of the algebra for the tree in Figure 7 are:

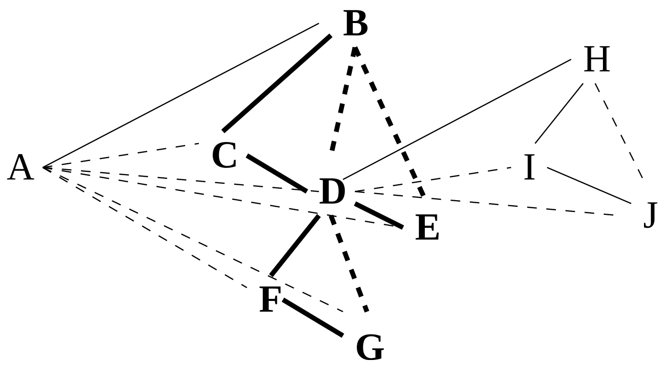

Figure 11: Local Yield Height Example

$F(\sim,\sim)$
$E(F(\sim,\sim),\sim)$
$D(\sim,\sim)$
$C(D(\sim,\sim),E(F(\sim,\sim),\sim))$
$B(C(D(\sim,\sim),E(F(\sim,\sim),\sim)),\sim)$
$A(\sim,B(C(D(\sim,\sim),E(F(\sim,\sim),\sim)),\sim))$

For a more complicated example, here are the terms of Figure 9:

$G(\sim,\sim,\sim)$
$F(G(\sim,\sim,\sim),\sim,\sim)$
$E(\sim,\sim,\sim)$
$J(\sim,\sim,\sim)$
$I(J(\sim,\sim,\sim),\sim,\sim)$
$H(\sim,I(J(\sim,\sim,\sim),\sim,\sim),\sim)$
$D(E(\sim,\sim,\sim),F(G(\sim,\sim,\sim),\sim,\sim),$
  $H(\sim,I(J(\sim,\sim,\sim),\sim,\sim),\sim))$
$C(D(E(\sim,\sim,\sim),F(G(\sim,\sim,\sim),\sim,\sim),$
  $H(\sim,I(J(\sim,\sim,\sim),\sim,\sim),\sim)),\sim,\sim)$
$B(\sim,C(D(E(\sim,\sim,\sim),F(G(\sim,\sim,\sim),\sim,\sim),$
  $H(\sim,I(J(\sim,\sim,\sim),\sim,\sim),\sim)),\sim,\sim),\sim)$
$A(\sim,\sim,B(\sim,C(D(E(\sim,\sim,\sim),F(G(\sim,\sim,\sim),\sim,\sim),$
  $H(\sim,I(J(\sim,\sim,\sim),\sim,\sim),\sim)),\sim,\sim),\sim))$

4 Heights

4.1 Local Yield Heights

It is useful to develop some sort of concept for the heights of these trees. We employ three such concepts. The *local yield height* is used to denote the i-dimensional height of a given i-dimensional local yield. Figure 11 depicts a 3-dimensional tree, with the 2-dimensional local yield rooted at B highlighted.

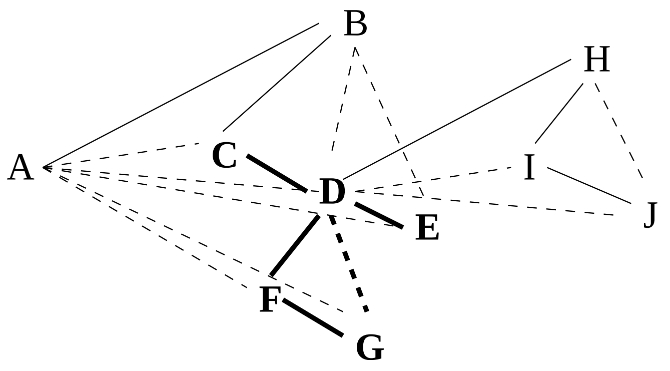

Figure 12: Forest Height Example

Note that the highlighted portion does not extend along the third dimension from D to the tree rooted in H, because that portion of the structure is 3-dimensional and is not part of the strictly 2-dimensional local yield. It should be immediately apparent from the figure that the 2-dimensional height of the local yield rooted in B, the highlighted portion, ought to be 2.

Definition 9 *Local Yield Height [Initial]*

- $h_{yield}(t, i) = -1$*, if* $t = \sim$
- $h_{yield}(t, i) = \max(h_{yield}(t_j, i)) + 1$ *where* t_j *ranges over* all *of* t*'s* i*-dimensional children, if* $t = T(X, t_1, t_2, \ldots, t_i, \ldots, t_d)$

Note that heights are also defined for i-dimensional subtrees of the local yield that are not properly local yields themselves. In the example, the 2-dimensional subtrees rooted at C, D, and E are assigned heights of 0, 1, and 0 respectively, even though none of these is a 2-dimensional local yield. B's height is then defined to be one greater than the maximum of the heights of its 2-dimensional children, namely the trees rooted at C, D, and E, which is equal to 2.

4.2 Forest Heights

The local yield height represents the i-dimensional height of an i-dimensional local yield, which is also a singleton i-dimensional forest, the nodes of which may be the roots of higher-dimensional trees. In contrast, we use the *forest height* to denote the height of a full i-dimensional forest. Figure 12 depicts the same 3-dimensional tree, this time with the 2-dimensional forest rooted at C highlighted. It can be seen that the forest height differs from the local yield height only in that our forest height definition must take into account the heights of all the trees in the forest, rather than just the height of the minimum tree in ordering of the forest. Our definition of forest height can therefore be simply stated as the maximum of the heights of the individual i-dimensional trees in the forest. However, we can also provide a recursive definition based only upon the forest heights of the root's immediate successors. This definition is attractive for implementation purposes, so it is the one we shall use.

Definition 10 *Forest Height*

- $h_{forest}(t, i) = -1,$ *if* $t = \sim$
- $h_{forest}(t, i) = \max(h_{forest}(t_i, i) + 1, \max_{1 \leq j < i}(h_{forest}(t_j, i))),$ *if* $t = T(X, t_1, t_2, \ldots, t_i, \ldots, t_d)$

The idea behind this definition is that the height of a forest is simply the maximum of all its immediately succeeding subforests in the dimensions of the forest, except that the height of the subforest in the maximum dimension (i) of the forest is increased by one for the purpose of counting that link. In the example, the height of the 2-dimensional forest rooted at D is defined as the maximum of the height of the forest rooted at E and one plus the height of the forest rooted at F; since these are both 0, the forest rooted at D has height 1. In turn, the 2-dimensional forest rooted at C is defined as the maximum of the height of the forest rooted at D and one plus the height of $\sim$, so the height of the forest rooted at C is also 1.

Our definition of forest height also permits us to provide a simpler definition of the local yield height that is based only upon the heights of the root's immediate successors, which again is attractive for implementation purposes. Our current definition of local yield height calls for the maximum height of any of the root's children. This turns out to be exactly the forest height of the root's minimum child, which is the maximum height of itself and all of its siblings.

Definition 11 *Local Yield Height [Final]*

- $h_{yield}(t, i) = -1,$ *if* $t = \sim$
- $h_{yield}(t, i) = h_{forest}(t_i, i) + 1,$ *if* $t = T(X, t_1, t_2, \ldots, t_i, \ldots, t_d)$

Since the height of the forest rooted at C is 1, the height of the local yield rooted at B again is equal to 2, as desired.

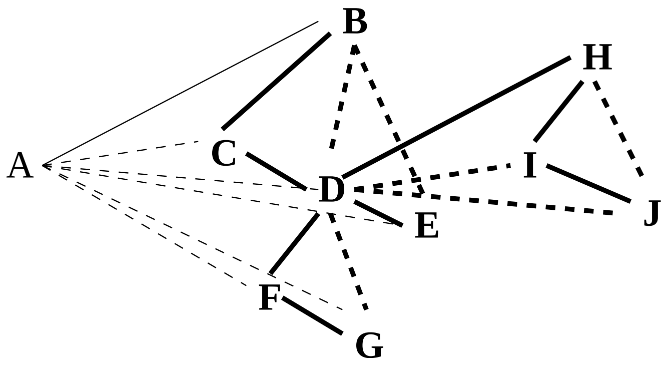

Figure 13: Child Structure Height Example

4.3 Heights of Child Structures

Aside to selves: we may eventually want to define a projection height, which would be the i-dimensional height of a d-dimensional structure after it has been projected down to i dimensions. This could be easily confused with child structure height, so we may want to think about dropping this subsection.

It would also be useful to define a height for the entire child structure rooted at a node. Figure 13 depicts the same tree again, with the entire child structure rooted at B highlighted. We may refer to the height of this child structure in any dimension. Its 1-dimensional height should be two, since the nodes labeled E, G, and J each require the use of two 1-dimensional links to reach from B. Similarly, its 2-dimensional height should be two, since the nodes F, G, I, and J each require the use of two 2-dimensional links to reach from B. Finally, its 3-dimensional height should be one, since each of the nodes H, I, and J require one 3-dimensional link to reach.

Definition 12 *Child Structure Height*

- $h_{child}(t, i) = -1$, *if* $t = \sim$
- $h_{child}(t, i) = \max(h_{child}(t_i, i) + 1, \max_{j \neq i}(h_{child}(t_j, i)))$, *if* $t = T(X, t_1, t_2, \ldots, t_i, \ldots, t_d)$

The logic of the child structure height definition is the same as that of the forest height, except that here we do not confine ourselves to the dimensions of an individual forest, but instead we consider the extents of the trees rooted in that forest as well.

5 Headed Trees

When determining the string yield of a tree in dimensions higher than 2, there is ambiguity in the way that the yield can be determined. To resolve this, multidimensional trees need to be marked with a "head" for each local yield. The head is a node in each local yield that indicates how adjoined local yields should be merged to produce the correct lower-dimensional yield. A "spine" can then be determined as the path through all the heads in the tree. In our diagrams, spines will be indicated by a double line.

Since the head of a local yield must be located along the spine, we can uniquely indicate the position of the head by giving its offset from the root along the spine. This offset must be an integer in the range 0–$h_{spine}(t, i)$, where $h_{spine}(t, i)$ denotes the spine length of the i-dimensional local yield t, which will be defined shortly in Definition 14. Using this approach, we can define a function $\textsc{Head}(n, t, i)$ that maps an i-dimensional lodcal yield t with a head offset n to the i-dimensional head of t.

Definition 13 $\textsc{Head}(n, t, i)$

- $\textsc{Head}(n, t, i) = t$, *if* $t = \sim$ *or* $n = 0$
- $\textsc{Head}(n, t, i) = \textsc{Head}(n - 1, \textsc{Head}(m, t_i, i - 1), i)$, *where* m *denotes the head offset of the* $(i - 1)$*-dimensional local yield* t_i,
  if $t = T(X, t_1, t_2, \ldots, t_i, \ldots, t_d)$

$\textsc{Head}$ is guaranteed to terminate because, for each recursion, either i is decremented by 1 and n is permitted to vary, or n is decremented by 1 and i is held constant. When n is zero, the base case holds and simply returns t. When i is zero, n must also be zero, because the head of a 0-dimensional local yield must be the root of the local yield—it can have no 0-dimensional successors. Because i never increases and because increasing n requires decreasing i, one of these cases must eventually be reached.

Each iteration of $\textsc{Head}$ first finds the head of the ith dimensional successor of t and then decrements the offset counter n. When the base case is reached and $n = 0$, the head of the ith dimensional successor has been followed n times. Since n is the offset of the head down the spine from t, the correct head has then been found.

5.1 Definitions

Definition 14 *i-dimensional Spine Length of a local yield t*

- $h_{spine}(t, i) = -1$, *if* $t = \sim$

- *$h_{spine}(t,i) = 0$, if $t = T(X, t_1, t_2, \ldots, t_i, \ldots, t_d)$ and t is the root of a j-dimensional local yield, where $j < i$*
- *$h_{spine}(t,i) = h_{spine}(t_i, i) + 1$, if $t = T(X, t_1, t_2, \ldots, t_i, \ldots, t_d)$ and t is the root of an i-dimensional local yield*
- *$h_{spine}(t,i) = h_{spine}(t_i, i)+1$, if $t = T(X, t_1, t_2, \ldots, t_i, \ldots, t_d)$ and t is the root of a j-dimensional local yield and $t = \textsc{Head}(n, t, j)$, where $j > i$ and n is the head offset of t*
- *$h_{spine}(t,i) = h_{spine}(\textsc{Head}(n, t, j), i)$, if $t = T(X, t_1, t_2, \ldots, t_i, \ldots, t_d)$ and t is the root of a j-dimensional local yield and $t \neq \textsc{Head}(n, t, j)$, where $j > i$ and n is the head offset of t*

Using the spine length as an offset limit, we can now mark the heads of the local yields by annotating the root t of each i-dimensional local yield with the offset of its i-dimensional head along the spine.

Definition 15 *Tree-ordered forests with heads*

- *$\sim$ is an (empty) (j,d)-forest for all $0 \leq j \leq d$.*
- *If t_1, t_2, …, t_d are, respectively, $(0,d)$-, $(1,d)$-, …, $(d-1,d)$-forests and $X \in \Sigma$, then $t = T(X, t_1, t_2, \ldots, t_d, n_1, n_2, \ldots, n_d)$ is a (j,d)-forest for all $0 \leq j \leq i$, where:*
  - * *i is the smallest dimension such that t_i is not empty or d if t_k is empty for all $1 \leq k \leq d$*
  - * *$0 \leq n_k \leq h_{spine}(t_k, k-1)$, for all $1 \leq k \leq d$ such that t_k is not empty.*
  - * *For those t_k that are empty, n_k is undefined.*
  - * *Each t_k is the successor of the new node labeled X in the kth dimensionwith n being the head offset in dimension i.*
- *Nothing else is a tree-ordered forest.*

5.1.1 Example

In the example tree in Figure 14, B is the root of a 2-dimensional local yield. The head of this local yield is E. To find the head from B, $\textsc{Head}(2, B, 2)$ is called, telling it to find the head of A's 2-dimensional local yield. Head would then call $\textsc{Head}(1, C, 1)$, recursively asking itself to first find the head of C's 1-dimensional local yield. To do this, Head again recursively calls

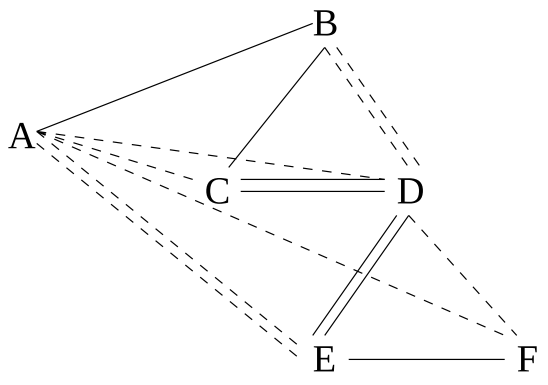

Figure 14: Example Headed Tree

$\textsc{Head}(0, D, 0)$, which returns that D is the head. Having found the 0-dimensional head of D's local yield, which, coincidentally, is also the 1-dimensional head of C's local yield, the $\textsc{Head}$ function continues from there, looking again for the 2-dimensional head and decrementing the supplied head offset by 1. The $\textsc{Head}$ function will continue to be called until the first call finally returns. The sequence of calls is shown below.

$$\begin{aligned}
&\textsc{Head}(2, B, 2)\\
&= \textsc{Head}(1, \textsc{Head}(1, C, 1), 2)\\
&= \textsc{Head}(1, \textsc{Head}(0, \textsc{Head}(0, D, 0), 1), 2)\\
&= \textsc{Head}(1, \textsc{Head}(0, D, 1), 2)\\
&= \textsc{Head}(1, D, 2)\\
&= \textsc{Head}(0, \textsc{Head}(0, E, 1), 2)\\
&= \textsc{Head}(0, E, 2)\\
&= E
\end{aligned}$$

5.2 Concrete Forms

We must amend our abstract data type and our flat form to include the locations of the heads. Since the head is a node in the tree, we will indicate it in the abstract data type definition simply by adding a pointer to it in the class instance for the root of the local yield:

```
public:
 forest(label_type label, vector<forest*> links, forest *new_head)
 ...
 forest* get_head();
 void set_head(forest* new_head);
 ...
```

```
private:
  ...
  forest* head;
  ...
```

Our flat form definition again follows immediately from the constructor.

Definition 16 *Flat Form with Heads*

- $\sim$ *represents an (empty)* (j, d)*-forest in flat form for all* $0 \leq j \leq d$.
- *If* t_1, t_2, …, t_d *are, respectively,* $(0, d)$-, $(1, d)$-, …, $(d-1, d)$*-forests in flat form and* $X \in \Sigma$, *then* $t = X(t_1[n_1], t_2[n_2], \ldots, t_d[n_d])$ *is a* (j, d)*-forest in flat form for all* $0 \leq j \leq i$, *where:*
  * i *is the smallest dimension such that* t_i *is not empty or* d *if* t_k *is empty for all* $1 \leq k \leq d$
  * $0 \leq n_k \leq h_{spine}(t_k, k-1)$, *for all* $1 \leq k \leq d$ *such that* t_k *is not empty.*
  * *For those* t_k *that are empty,* n_k *is undefined.*
  * t_k *is the successor of the new node labeled* X *in the* k*th dimension, and* n *is the head offset in dimension* i.
- *Nothing else is a tree-ordered forest in flat form.*

5.2.1 Example

Looking back at the tree in Figure 14, the flat form with the heads included would be written

$$A(\sim, \sim, B[2](\sim, C[1](D[0](\sim, E[0](F[0](\sim, \sim, \sim), \sim, \sim), \sim), \sim, \sim), \sim))$$

6 Embed

It is useful for our parsing algorithm to be able to handle arbitrary-branching trees of arbitrary dimensions as well as the 2-branching trees of arbitrary dimensions that it already handles. The intuitive approach would be to use the tree data structure to represent the local trees instead of a list of states. However, we would prefer to adapt our existing code that works with 2-branching structures, so we will use an algorithm to embed, or factor, an arbitrary branching rule into a set of 2-branching rules. But first, a lemma about 2-branching trees.

Lemma 2 *Every n-dimensional successor (in the left-child, right-sibling sense) in a d-dimensional, 2-branching tree can have only an $(n-1)$-dimensional successor and a d-dimensional successor.*

Proof: An n-dimensional successor in a d-dimensional, 2-branching tree cannot have any successor in dimensions less than $n-1$ by Lemma 1, and it can have no successor in any dimension greater than or equal to n, except for dimension d, because the node already has a parent in each such dimension, and so the inclusion of such a successor would make the tree more than 2-branching. $\dashv$

Corollary 2 *In a d-dimensional 2-branching local tree there are exactly $d+1$ nodes.*

Since no element in the yield of the local tree will have a successor in the d-dimension, and the root will only have a successor in the d-dimension, each node in the local tree only has one successor.

6.1 Size of Trees

This leaves open a question of the size of trees in higher branching factors. As seen in Figure 15, the number of nodes in a 3-dimensional, 3-branching tree (14 nodes) is large, compared to the 2-dimensional 3-branching tree (4 nodes). We can look at the number of nodes in these structures and determine how fast they grow as the dimensionality increases. Looking at a d-dimensional 2-branching structure, Corollary 2 requires there to be $d+1$ nodes in the structure. This growth is linear.[4]

For 3-branching local trees, the number of nodes in d-dimensions ,$N_3(d)$, can be stated as $N_3(d) = (N_3(d-1)-1)N_3(d-1)+2$ or simplified as $N_3(d) = N_3(d-1)^2 - N_3(d-1)+2$. This equation comes from the way these trees grow as the dimension increases. Looking at Figure 15, the 3-dimensional tree can be seen as a full 3-branching 2-dimensional local tree with another full 3-branching 2-dimensional tree attached at every node, except the root. So we are multiplying the number of nodes in a $(d-1)$-dimensional structure, $N_3(d-1)$, by the number of nodes in the original $(d-1)$-dimensional yield minus the root, $N_3(d-1)-1$. Then we have to add the root back in, plus a new root to anchor the tree in the d-dimension.

[4]Another perspective on the growth of 2-branching trees is to look at them as simplexes. When we increase the dimensionality of a tree, we are simply adding a root in an orthogonal dimension to an existing local tree.

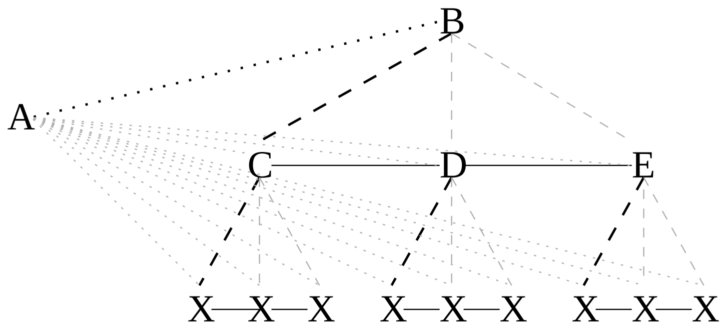

Figure 15: A full 3-dimensional, 3-branching tree.

We are interested in the lower bound on the size of these trees, so when we solve this recurrence we only need to look at the largest term. The first step in the recursion is

$$N_3(d) = (N_3(d-2)^2 - N_3(d-2) + 2)^2 - (N_3(d-2)^2 - N_3(d-2) + 2) + 2$$

The largest term is $(N_3(d-2)^2)^2$. If we carry the recursion out to the next step, this term grows to $(N_3((d-2)^2)^2)^2$. In general, the entire term is squared for every step in the recursion, of which there are d. So we can say that the number of nodes in a 3-branching n-dimensional tree is $\Omega(k^{(2^{(d-1)})})$ for some k. Moving to 3-branching, then, means an abrupt shift from linear size to hyper-exponential.

6.2 Embed Algorithm

To allow our parsing algorithm to use simpler structures, we would like to work with only 2-branching trees. To accomplish this, we will use an algorithm to factor these n-branching structures into sets of 2-branching local trees, similar to the CNF transformation.

To factor an arbitrary branching d-dimensional local tree into a set of 2-branching d-dimensional local trees, we can simply traverse the local tree checking for illegal links. Since the factoring depends on the order in which we handle the links, we will standardize the process by always looking for illegal links in the higher dimensions first. When such links are found, we replace the parent node of the link with a new, unique label,[5] and we introduce a new rule rooted in that new label. The tree for the new rule consists of

[5]As a convention, if the original label was X, we will make the new label X'. This is simply a shorthand. It is important that X' be unique in the grammar as a whole to prevent the introduction of ambiguity. If this node X' already exists, we will simply append another prime marker until a unique label is formed.

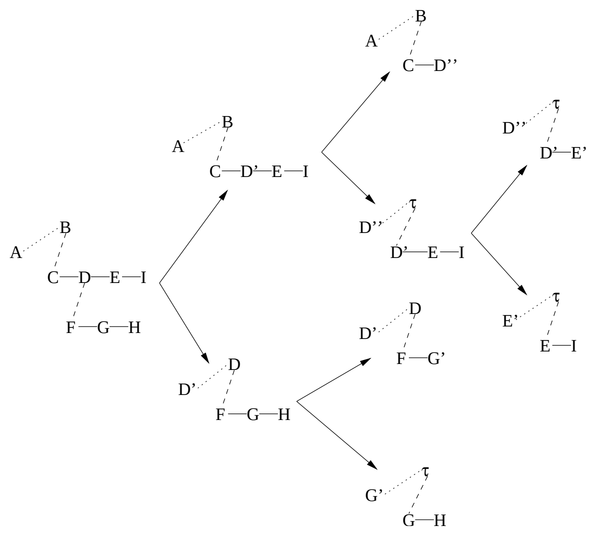

Figure 16: Example factoring of a 3d tree.

the original parent node of the link along with its i-dimensional yield, with i being the dimension of the illegal link. Figure 16 shows the progression from left to right of an arbitrary branching 3-dimensional structure being factored into 2-branching structures. In the first factoring, the node D is found to have an illegal link in the second dimension (it also has one in the first dimension, but this will be handled later). The resulting two structures are shown.

If the dimension of an illegal link is less than $n-1$, then simply adding the new label as the root will create a lower dimensional structure. To promote the structure to the proper dimension, nodes are added with a special value which we will call τ. This can be seen in the structures rooted at G', E', and D'' in Figure 16. Since these nodes are added by the factoring process, there is no circumstance in which a rule will be rooted at τ. The only rules generated by the factoring process are rules that already existed, or rules rooted at one of the unique labels created. Since τ has this restriction, we can give all such nodes the same label, without having to distinguish one from the other. A τ node is not part of the image of the tree but is intended to help us "excavate" or reconstruct the arbitrary-branching structure after parsing is complete.

Because τ is never the root and occurs only in 2-branching rules, we can guarantee that each τ node will always have exactly one successor. Figure 17 shows a 4-dimensional rule being factored into two 2-branching rules. Notice how τ is used to allow E' to be the root of a full rule containing only E and F. Without the use of τ, the rule would only be 2-dimensional. Because of

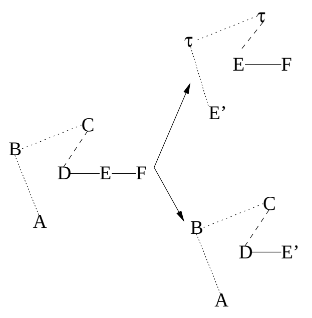

Figure 17: A 4-dimensional local tree of a grammar being factored into 2-branching local trees.

this use of chaining τ nodes, there will never be more than one successor for any such node.

Here is a more detailed explanation of the factoring shown in Figure 16. The first illegal link encountered is the second dimensional successor of D. D is re-labeled as D' and the illegal yield is removed. This result is shown in the top branch in the figure. A new tree is formed with its root labeled D' and the illegal yield is set as the yield of the new tree. This new tree is shown in the lower branch in the figure. Now the two new trees are processed. The upper tree finds another illegal link, the 1-dimensional link on D'. As before, D' is re-labeled D'' and the illegal yield is chopped off (upper branch). A new tree with a root labeled D'' is formed, similar to before. This time, however, we cannot just set the illegal yield as the yield of the new tree, because we only have a 1-dimensional yield, while a 2-dimensional yield is required. This is where τ comes in. To create a 2-dimensional yield, we will set the 1-dimensional yield as the successor of a τ node. Now we have a 2-dimensional yield that can be used (lower branch). This process continues until trees with no illegal nodes are left. In this example, the tree is factored into five 2-branching trees.

6.3 Algorithm

```
;embed takes the label of the root of a
; local tree in a grammar, the local
; yield of that root, and a grammar;
; and adds the rule to the grammar
; in embedded form.
embed(label,(n-1)-dimensional tree t,
      grammar) {

  new tree startTree = t
```

```
for dim from n-1 to 0 {
  ;Loop Invariant: There are no illegal
  ; links in the nodes above t in
  ; startTree.

  for i from dim+1 to n {
   ;Loop Invariant: t has no illegal
   ; links in all dimensions from dim+1
   ; to i.

    ;At this point, an i-dimensional
    ; link is illegal.
    if (t has link i) {
      new tree u
      relabel t with unique label

      for j from 1 to n-1 {
        ;Loop invariant: u has no
        ; successor in any dimension
        ; less than j, except for an
        ; i-dimensional successor.

        if ( j != i and
            u has no j-dimensional
              link )
          delete j-dimensional link of u

      ;filltree extends u into a d-dimen-
      ; sional tree by adding tau.
      u = filltree(t.link(i), i,
                   automaton)
      ;Now we want to keep looking for
      ; illegal links
      embed(unique label T', u, t')
      ;We can finally remove the
      ; illegal link
      t.unlink(i)
    }
    ;Postcondition: u has no successor
    ; except for an i-dimensional
    ; successors.
  }
  ;Postcondition: t has no illegal links
  ; in any dimension

  ;All illegal links have been fixed, so
  ; move to the next node.
  t = t.link(dim) if dim > 0
  }
  ;Postcondition:  There are no illegal
  ; links anywhere in startTree.

}

;filltree takes a tree, the dimension of
; the tree, and a grammar and adds
; roots to t so that it is now a complete
; grammar rule.
filltree(n-dim tree t, dim, grammar) {

  for i from dim to n-1 {
    ;Loop invariant:  t is an i-dimensional
    ; local yield.

    if ( i = t->dim() )
       new tree newtree is labeled tprime
    otherwise
       new tree newtree has label of tau
       create empty tree with label
         of tau
       add that tree to the grammar
    newtree->set_link(i,t)
  }
  ;Postcondition: t is an (n-1)-dimensional
  ; local yield.

  return newtree
}
```

6.4 Growth in Grammar Size

While it does allow us to parse with simpler structures, the factoring process cannot improve the hyper-exponential size of the grammar as a function of the dimensionality. Since every node in the n-branching structure must be represented in the grammar, the growth in the size of the grammar must coincide with the growth in the size of n-branching trees. We can see this by looking at the illegal links present in a 3-branching local tree. Remember that we consider all links that are not in a 2-branching local tree but are in an n-branching local tree illegal. This means that the number of illegal links in a given structure can be defined as $N_k(d) - N_2(d)$, where k is the branching factor of the tree, since all the links in the 2-branching tree must also be in the k-branching tree and no other links in the k-branching tree can be legal.

The *embed* function is designed to fix one illegal link for every new local tree created. There is a side-effect, however, when a higher dimensional

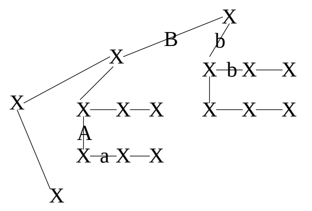

Figure 18: When one of the links labeled with a capital letter is fixed, the links labeled with the same letter in lowercase will also be fixed.

illegal link is fixed. In a full tree, every n-dimensional illegal link fixed also fixes an i-dimensional illegal link for all $i < n$. This can be seen in Figure 18. When the 2-dimensional link labeled "A" is fixed, the link labeled "a" will also be fixed as well. When the structure is moved to a new local tree to fix "A", "a" will automatically move to a 2-branching position, so no new local tree needs to be created to fix it. Similarly, when the illegal 3-dimensional link labeled "B" is fixed, the two links labeled "b" will be also be fixed.

Looking at a full 3-dimensional 3-branching tree, there are 3 illegal 2-dimensional links and 7 illegal 1-dimensional links. Since each time we fix a 2-dimensional link, a 1-dimensional link will also be fixed, we only need to create new rules to fix 4 1-dimensional links ($7 - 3 = 4$). In total, we will create 7 new rules, 3 to fix 2-dimensional links and 4 to fix 1-dimensional links. In the 4-dimensional case, let there be x 3-dimensional illegal links, y 2-dimensional illegal links, and z 1-dimensional illegal links. We will fix x 3-dimensional links, $y - x$ 2-dimensional links, and $z - x - (y - x)$ 1-dimensional links. Let *fix*(d) be the number of dimensions that need to be fixed in dimension d.

$$\begin{aligned}&\mathit{Fix}(3) = x\\&\mathit{Fix}(2) = y - x\\&\mathit{Fix}(1) = z - x - (y - x) = z - y\end{aligned}$$

Notice that when we add the terms together to get the total number of links to be fixed, everything cancels except for z, the number of 1-dimensional illegal links. In fact, the number of links to be fixed is always equal to the number of 1-dimensional illegal links.

Theorem 1 *For a full d-dimensional, n-branching local tree, the number of local trees in the factored form required is equal to the number of 1-dimensional illegal links.*

Before we can prove this, we need to establish some lemmas.

Lemma 3 *In a full n-branching tree, when an i-dimensional link is fixed using embed, a j-dimensional link is also fixed for all $0 < j < i$.*

Proof: *Embed* will set the i-dimensional link to the i-dimensional position in a 2-branching tree. Since the i-dimensional link is now a legal link, it is allowed to have an $(i-1)$-dimensional successor by Lemma 2. In a full n-branching tree, it is guaranteed to have such a link. Since that $(i-1)$-dimensional successor is now also a legal link, it is allowed to have an $(i-2)$-dimensional link, which it is guaranteed to have in a full n-branching tree. This continues to the minimal dimension. $\dashv$

Lemma 4 *Let $l(i)$ be the number of i-dimensional links that are not fixed by the side-effect of fixing higher dimensional links as in Lemma 3 and $t(i)$ be the total number of i-dimensional illegal links, then $l(x) = t(x) - t(x+1)$.*

Proof: (by induction on $d - x$)
Base Case: $l(d-1) = t(d-1)$ (By Lemma 3).
Inductive Hypothesis: For inductive purposes, assume that $l(d-j) = t(d-j) - t(d-j-1)$ for all $1 < j < d-x$.
Induction:

$$\begin{aligned}
& l(x) \\
&= t(x) - \sum_{x<i<d} [l(i)] \\
&= t(x) - \sum_{x<i<d-1} [l(i)] + l(d-1) \\
&= t(x) - \sum_{x<i<d-2} [l(i)] + \\
&\qquad l(d-2) + t(d-1) \\
&\qquad\qquad \text{(By base case)} \\
&= t(x) - \sum_{x<i<d-2} [l(i)] + \\
&\qquad (t(d-2) - t(d-1)) + t(d-1) \\
&\qquad\qquad \text{(By IH)} \\
&= t(x) - \sum_{x<i<d-2} [l(i)] + t(d-2) \\
& \text{repeating for upper bound } d-3 \ldots x+1 \\
&= t(x) - \sum_{x<i<x+2} [l(i)] + t(x+2) \\
&= t(x) - (t(x+1) - t(x+2)) + \\
&\qquad t(x+2) \\
&= t(x) - t(x+1)
\end{aligned}$$

⊣

Proof: Proof of Theorem 1
Let the total number of local trees in factored form required be equal to

$$l(1) + l(2) + \ldots + l(d-1)$$

or the sum of all illegal links not fixed by Lemma 3. Using Lemma 4,

$$\begin{aligned} l(1) + l(2) + \ldots + l(d-1) = \\ &(t(1) - t(2)) + (t(2) - t(3)) + \\ &(t(3) - t(4)) + \ldots + \\ &(t(d-2) - t(d-1]) + t(d-1) \\ &= t(1). \end{aligned}$$

⊣

Theorem 1 is not true for trees without the maximum number of illegal links. It is the worst case, however, since the side-effect described in Lemma 3 will only fail to fix a lower dimensional link if it does not exist, and therefore does not require a new factored tree anyway.

Even though Theorem 1 proves that we do not need a new grammar rule for every illegal link in the tree, the growth of the grammar will still be hyper-exponential in the dimension of the local trees because the number of 1-dimensional links is a constant fraction of the number of total links. In the worst case (a full tree), every 2-dimensional successor will be the root of a 1-dimensional local tree containing $n-1$ 1-dimensional links, where n is the branching factor. Similarly, every 3-dimensional successor will be the root of a 2-dimensional local tree containing $n-1$ 2-dimensional links. In general, every d-dimensional successor will be the root of a $(d-1)$-dimensional local tree containing $n-1$ $(d-1)$-dimensional links.

This is not worse than is minimally necessary to accomodate every node, however, since the number of new trees is asymptotic to the number of nodes in the original tree.

6.5 Excavation

We will call the process to recover the arbitrary branching structures from the 2-branching local trees “excavation”. In a top-down traversal of the tree, every τ node will be replaced with its only successor, and each unique label created during the factoring will be replaced with its n-dimensional successor.

We can recover the original heads by adding together the i-dimensional head offsets of the adjoined structures, where i is the dimension of the link that was originally removed in the embedding process. The sum will be the new i-dimensional head offset of the adjoined structure, which will be the same head offset used in the the arbitrary-branching rule that produced the adjoined structure. If we are careful not to leave a node until it does not contain a τ or a unique label (this may require many operations on one node), we will have the original structure when the traversal has terminated.

7 Project

We want to be able to transform a tree of any dimension into a lower dimensional tree. This will allow us to see the string yield of higher dimensional trees, and also let us look at complex trees in a form that is easier to grasp. We will use an algorithm that takes an n-dimensional tree and returns the string yield of that tree. This can be done with a recursive traversal through the tree starting at the root and working towards the leaves, reducing a d-dimensional tree to a $(d-1)$-dimensional tree. This process can be repeated until a tree of the desired dimensionality is formed.

This is done conceptually by traversing through the structure and replacing each node that has a d-dimensional successor with the structure rooted at that successor, thus eliminating the d-dimensional link. If there is a $(d-1)$-dimensional successor of the removed node it will be made a $(d-1)$-dimensional successor of the maximal node (foot) in the $(d-1)$-dimensional spine of the new structure. This maximal node will not already have a $(d-1)$-dimensional successor, since having one would not make it maximal. Since we are simply extending the length of the spine by doing this, we can be assured that this successor is allowed to be moved here.

All other successors (less than $(d-1)$-dimensional) of the original node will become successors of the node that replaced it. Since the new node was a d-dimensional successor, it is the root of a $(d-1)$-dimensional yield, and therefore cannot have any succesors in dimensions less than $d-1$. Thus, we are guaranteed to have no conflicts when setting the successors of the original node as successors of the new node. We can verify that these successors are allowd to become successors of the new node by considering that they were legal successors of the original node, and the new node is now in the same place as the original node.

In Figure 19 a 3-dimensional tree is being projected to a 2-dimensional tree. First, The 3-dimensional link between A and B is removed by replacing A with the structure rooted at B. Since A has no other successors, this has

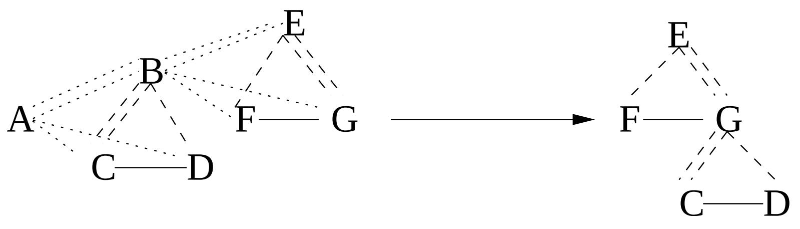

Figure 19: A 3d tree being projected to a 2d tree.

the same effect as simply removing A from the tree. Next, the 3-dimensional link from B to E will be removed. This will be done the same way, by replacing B with the structure rooted at E. This time, however, B has a successor in the second dimension. Since it is a 2-dimensional successor, it will become the 2-dimensional successor of the head of E's 2-dimensional yield, G.

7.1 Implementation

```
;project takes a tree, 'tree', and the dimension of the tree,
;  'n', and returns a tree of dimension 'n-1' with the same
;  yield as 'tree'.
project(tree,n)
  ;If the tree is empty, then nothing needs to be done.
  if(tree == empty)
    return tree;

  ;If the tree has an n-dimensional link, it must be replaced
  ;  with the (n-1)-dimensional yield.
  if(tree has an nd link)
    ;The n-1-dimensional local tree must be attached at the
    ;  head of the (n-1)-dimensional local yield.
    if(n > 0 and tree has (n-1)-dimensional link)
      find_foot(tree.link(n)).link(n-1)=tree.link(n-1)
    ;Loop Invariant: The successors of tree's n-dimensional
    ;  successor from n-2 to i are now also the successors of
    ;  tree in the same dimension they were successors of
    ;  tree's n-dimensional successor.
    ;Postcondition: All the successors of tree's n-dimensional
    ;  successor are now successors of tree.
    for i from n-2 to 1
      tree.link(n).link(i) = tree.link(i)
    ;Recursively project
    tmp = project(tree.link(n))
    ;Erase the old fragment
    set all tree links to NULL
    delete tree
    ;Adjust dimensionality of new tree and return it
    tmp.set_dim(n-1)
    return tmp
  else
    ;Loop Invariant: All the successors from the 1st dimension
    ;  to the ith dimension have been projected to n-1
    ;  dimensions.
    ;Postcondition: All the successors in all dimensions have
    ;  been projected to n-1 dimensions.
    for i from 1 to n-1
      tree.link(i) = project(tree.link(i))
    ;Adjust dimensionality of tree and return it
    tree.set_dim(n-1)
    return tree
```

This algorithm can then be run on its own output, smashing a tree down by one dimension each time until a flat tree is returned.

8 Tree Recognizer

The tree recognizer determines whether a tree is licensed by a particular automaton. Each node in the tree is initially labeled with some element from Σ, the alphabet of the grammar. Each node in the tree is also assigned a state by the recognizer. The state of every node starts uninitialized. The recognizer begins at the root of the tree and takes the root's label and the states of the root's local yield. From this information, it is able to lookup the corresponding rule in the automaton and assign a new state to the root. If a node in the local yield is found to be uninitialized, the recognizer then recurs to that node. Since all states begin uninitialized, the recognizer recurs down the tree until a node with no successor is found. At this point that node is given a state based on a terminal rule from the automaton. In general when assigning the state to a particular node in the tree, the recognizer will try to find a local tree in the automaton which matches the root label as well as the states of it's local yield. It will assign a state to that node corresponding to

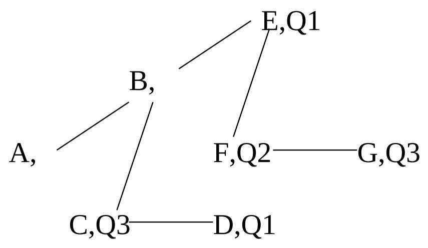

Figure 20: Tree with terminals labeled with states from automaton

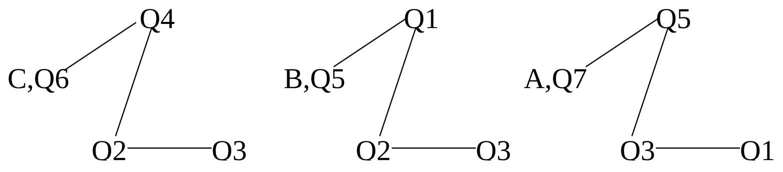

Figure 21: Automaton recognizing tree in Figure 20

the state of the root of the local tree found in the automaton. If at the end of this processes the state of the root of the entire tree being recognized is an accepting state, then the tree is accepted as being licensed by the automaton.

The tree in Figure 20 has already had states assigned to the leaf nodes from terminal local trees in the automaton. These states are the Q values following the Σ labels in the tree. Next, B will be assigned a state based on the second local tree of the automaton in Figure 21. This will be based on a function from Σ label B and the local tree Q1, Q2, Q3 to the state Q5. Next A will be assigned a state Q7 based on the third automaton rule.

8.1 Algorithm

```
;rec takes a sigma labeled tree and an automaton.
;  It returns the state of the root of the tree.
;  The tree will be modified to contain the states of the nodes.
rec(root, autmaton)

   ;recur on each valid link in root
   for i from 0 to dimension of tree
      rec(root.link(i),aut);

   root.set_state( aut.nextstate(root.label(), root.link(root.dim()-1)) )

   return root.state()
```

References

[BS91] William A. Baldwin and George O. Strawn. Multidimensional trees. *Theoretical Computer Science*, 84:293–311, 1991.

[Dew74] A. K. Dewdney. Higher-dimensional tree structures. *Journal of Combinatorial Theory*, 17:160–167, 1974.

[GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[JS96] Aravind K. Joshi and Yves Schabes. Tree-adjoining grammars. In A. Salomaa and S. Rozenberg, editors, *Handbook of Formal Languages and Automata*. Springer-Verlag, 1996.

[Rog03] James Rogers. wMSO theories as grammar formalisms. *Theoretical Computer Science*, 293(2):291–320, 2003.

[Sed98] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley, 3rd edition, 1998.

[Sip01] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 20 Park Plaza, Boston, MA 02116, 2001.

[Wei92] David J. Weir. A geometric hierarchy beyond context-free languages. *Theoretical Computer Science*, 104, 1992.